CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r. 82/08)

Canada

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Division
Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

0-315-05956-7

53876

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

Murray Scott Campbell

Date of Birth — Date de naissance	Country of Birth — Lieu de naissance
20 July, 1957	

Permanent Address — Résidence fixe

10404 135 st
Edmonton, Alta.
T5N 2C6

Title of Thesis — Titre de la thèse

Algorithms for the Parallel Search of Game Trees

University — Université

University of Alberta

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

M. Sc.

Year this degree conferred — Année d'obtention de ce grade	Name of Supervisor — Nom du directeur de thèse
1981	T. A. Marsland

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date	Signature
July 20, 1981	M. Campbell

NL-91 (4/77)

THE UNIVERSITY OF ALBERTA

Algorithms for the Parallel Search of Game Trees

by

© M. Campbell

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF Master of Science

Computing Science

EDMONTON, ALBERTA

Fall 1981

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR M. Campbell

TITLE OF THESIS Algorithms for the Parallel Search of Game Trees

DEGREE FOR WHICH THESIS WAS PRESENTED Master of Science

YEAR THIS DEGREE GRANTED Fall 1981

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(SIGNED) M. Campbell

PERMANENT ADDRESS:

10404 135 st.

Edmonton, Alta.

T5N 2C6

DATED July 20 1981

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled Algorithms for the Parallel Search of Game Trees submitted by M. Campbell in partial fulfilment of the requirements for the degree of Master of Science.

.....T. A. Marsland.....

Supervisor

.....

.....L. K. Schubert.....

.....J. W. D. Chinneck.....

Date.....20 June '81.....

Abstract

A comparison of algorithms for sequential and parallel search of game trees is presented. A basis for comparison is described which provides measures of algorithm performance on cases of theoretical and practical interest.

Current sequential search methods are examined and extended, thus providing a foundation for the development of parallel search algorithms. The parallel algorithms are designed in such a way as to reduce the problems involved in an actual multi-processor implementation. Simulated concurrency is used to compare the parallel algorithms proposed.

Acknowledgment

I would like to thank my supervisor, Dr. T.A. Marsland, for his advice and assistance throughout the development of this thesis. The continual encouragement he provided made my work very much easier.

Also thanks to Dr. L. Schubert, who made a number of useful suggestions.

Finally, the financial assistance received from NSERC is gratefully acknowledged.

Table of Contents

Chapter	Page
1. Introduction	1
2. Sequential Tree Searching Algorithms	3
2.1 Algorithm Descriptions	3
2.2 Performance Comparison	18
3. Approaches to Parallel Tree Search	31
3.1 Parallelism in Primitive Operations	31
3.2 Parallel Aspiration Searching	31
3.3 Tree Decomposition	33
4. Algorithms for Parallel Search	35
4.1 Tree-splitting	37
4.2 Principal Variation Techniques	43
4.3 SSS* Adaptations	48
5. Performance Comparison of Parallel Algorithms	50
6. Conclusion	57
6.1 Summary of Results	57
6.2 Topics for Future Research	58
References	60
Appendix 1 - Enhancements to Parallel Search	62
Appendix 2 - Tree Generation Methods	70
Appendix 3 - The SSS* Algorithm	73

LIST OF TABLES

Table		Page
1.	Simulation results, tree width=8, depth=2	24
2.	Simulation results, tree width=16, depth=2	24
3.	Simulation results, tree width=24, depth=2	24
4.	Simulation results, tree width=8, depth=4	26
5.	Simulation results, tree width=16, depth=4	26
6.	Simulation results, tree width=24, depth=4	27
7.	Simulation results, tree width=8, depth=6	27
8.	Search times, randomly ordered trees	52
9.	Search times, strongly ordered trees	53
10.	Search times, optimally ordered trees	54
11.	The G Operator	75
12.	Modified cases of the G Operator	81

LIST OF FIGURES

Figure		Page
1.	Minimax	5
2.	Negamax	5
3.	An example of a cutoff	6
4.	Alpha-beta	7
5.	Falphabeta	11
6.	Lalphabeta	11
7.	Palphabeta	13
8.	SCOUT	13
9.	TEST	15
10.	Treesplit	39
11.	UPDATE	40
12.	Pvsplit	46
13.	Transposition table access and management	66
14.	Gen	71
15.	Tree to demonstrate SSS*	76
16.	OPEN list for tree of Figure 15	76
17.	Tree for aspiration SSS*	78
18.	OPEN list for tree of Figure 17	78
19.	Tree illustrating non-dominance of SSS*	79
20.	OPEN list for tree of Figure 19	79

1. Introduction

Many current game-playing programs build and carry out searches on large trees of possible move sequences. In games like chess it has become clear in recent years that increasing the depth of the tree searched can vastly improve the playing ability of a given program [THOM81]. A logical method of improving searching speed is the utilization of multi-processing technology. The application of parallelism to game tree search is non-trivial however, primarily due to the inherently sequential nature of the most popular search method, the alpha-beta algorithm.

The main purpose of this thesis is to present and compare various algorithms for the parallel search of game trees. Towards this end, a number of sequential tree searching algorithms are reviewed, and empirical studies of their performance are made. The algorithms are then examined for adaptability to a parallel environment. The resulting search methods are compared using simulations of multi-processor systems. The method used to compare algorithms involves the generation of a number of independent trees with certain desired properties.

The type of trees considered in this thesis are those of games such as chess, which are classified as two-person, zero-sum games of perfect information. Given a position p in such a game, it is possible to represent all the potential continuations from p in the form of a *game tree*. The nodes

of the tree correspond to game positions, while the branches represent the moves. The leaves of a game tree are called *terminal* nodes, and are assigned a value by a *static evaluation function*. All the remaining nodes are classified as *non-terminal*. The task in searching a game tree is to determine the *minimax* value of the *root* node. Intuitively, the minimax value of a node is the best score achievable from that node against an opponent who similarly chooses his best moves. This concept will be formalized in Chapter 2.

Some further terminology associated with game trees: A node is at *depth k* if it is k moves, or k *ply*, from the root. The number of branches leaving any particular non-terminal nodes is the *branching factor* of that node. A *uniform game tree* is one in which all non-terminal nodes have the same branching factor, and all terminal nodes are at the same depth in the tree.

2. Sequential Tree Searching Algorithms

2.1 Algorithm Descriptions

The **minimax algorithm** assumes there are two players called Max and Min, and assigns a value to every node in a game tree (and in particular to the root) as follows: Terminal nodes are assigned static values that represent the desirability of the position from Max's point of view. Non-terminal nodes can be given minimax value recursively. If a non-terminal node p has Max to move, then the value of p is the maximum over the values of the successors of p. Similarly, if Min is to move he will choose the minimum over the values of the successors of p. Figure 1 gives the minimax algorithm in a C-like language. The following procedures are assumed to exist:

1. terminal(p) - returns true if p is a terminal position.
2. staticvalue(p) - returns an integer measuring the relative goodness of p.
3. generate(p) - generates all the successors of p and returns the number of successors.

The **negamax algorithm** is a variant of the minimax procedure which is often more convenient to use. In the negamax approach, the terminal nodes are assigned static values from the point of view of the side to move. This allows the value of non-terminal positions to be calculated

as the maximum of the negatives of the values of their successors, i.e. the negamax algorithm applies the same operator at all levels in the tree.

The negamax approach avoids having separate cases for Max to move and Min to move (this is done implicitly in the procedure staticvalue), and will be used throughout this thesis. See Figure 2 for the negamax algorithm.

The **alpha-beta algorithm** is able to evaluate a game tree at reduced cost by ignoring subtrees that cannot affect the final value of the root node. Such subtrees are said to have been *cutoff*. Figure 3 gives an example of a possible cutoff [BAUD78]. The circle nodes have been fully evaluated, while the others are awaiting final evaluation. From the definition of negamax, $v = \max\{3, -x\}$ and $x = \max\{-2, \ldots\}$. Thus $x \geq -2 \Rightarrow 2 \geq -x$. Since $3 > 2 \geq -x$, the value of v cannot be affected by examining the remaining successors of x, and therefore a cutoff can occur at x. Figure 4, implements this idea, maintaining two bounds for even and odd levels of the tree against which cutoffs can be made.

The interval enclosed by (alpha,beta) is referred to as the alpha-beta *window*. For the alpha-beta algorithm to be effective, the minimax score of the root node must lie within the initial window. This can be ensured by setting the window to (-INFINITY,+INFINITY). Generally speaking, however, the narrower the initial window, the better the

```
minimax(position p)
{   int m, i, t, w;
    if (terminal(p))          /*  p is a terminal node  */
        return(staticvalue(p));
                              /*                        */
    w = generate(p);          /*  determine successors  */
                              /*      p.1,...,p.w       */
    if (MaxToMove(p))
    {   m = -INFINITY;
        for i = 1 to w do
        {   t = minimax(p.i);
            if (t > m)  m = t;
        }
    }
    else                              /*  Min to move  */
    {   m = +INFINITY;
        for i = 1 to w do
        {   t = minimax(p.i);
            if (t < m)  m = t;
        }
    }
    return(m);
}
```

Figure 1: Minimax

```
negamax(position p)
{   int m, i, t, w;
    if (terminal(p))
        return(staticvalue(p));
                              /*                        */
    w = generate(p);          /*  determine successors  */
                              /*      p.1,...,p.w       */
    m = -INFINITY;
    for i = 1 to w do
    {   t = -negamax(p.i);
        if (t > m)  m = t;
    }
    return(m);
}
```

Figure 2: Negamax

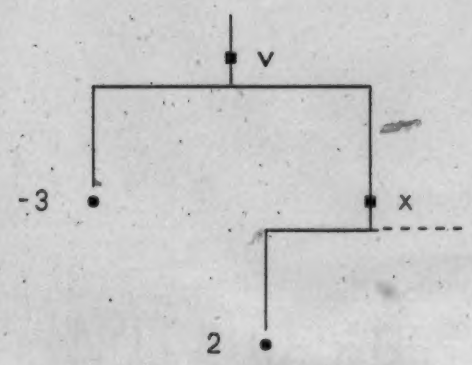

Figure 3: An example of a cutoff

```
alphabeta(position p, int α, int β)
{   int m, i, t, w;
    if (terminal(p))          /*  p is a terminal node  */
        return(staticvalue(p));
                              /*                        */
    w = generate(p);          /*  determine successors  */
                              /*      p.1,...,p.w       */
    m = α;
    for i = 1 to w do
    {   t = -alphabeta(p.i, -β, -m);
        if (t > m)  m = t;
        if (m ≥ β)  return(m);  /* cutoff */
    }
    return(m);
}
```

Figure 4: Alpha-beta

algorithm's performance. This provides the motivation for *aspiration searching*, in which the window is initialized to (V-e,V+e), where V is an estimate of the minimax value and e the expected error.

There are three possible outcomes of an aspiration search on a position p, depending on S, the minimax score of p.

1. if $S \leq V-e$, alphabeta(p,V-e,V+e) $\leq$ V-e
2. if $S \geq V+e$, alphabeta(p,V-e,V+e) $\geq$ V+e
3. if $V-e < S < V+e$, alphabeta(p,V-e,V+e) = S

Cases 1 and 2 are called *failing low* and *failing high* respectively [FISH80]. Only in case 3 is the true score of p found. Searches that fail high or low must be repeated with a window that actually encloses S.

The aspiration search concept has spawned a number of variants on alpha-beta that attempt to employ the technique to improve search speed. Some of these alpha-beta modifications can profit from **falphabeta**, for "fail-soft-alphabeta" [FISH80]. It has been noted that, in case of a failed aspiration search, the search must be repeated with a more realistic window. If a search with window (V-e,V+e) fails high, for example, the window (V+e,+INFINITY) is guaranteed to find the true score. Falphabeta can sometimes return a tighter bound on the score of the tree, and the second search can use this bound to

advantage. Figure 5 illustrates falphabeta.

There are two differences between alphabeta and falphabeta. Initially m is assigned to -INFINITY instead of alpha, and the third parameter of the recursive call to falphabeta must therefore become -max(m, alpha). It has been proven [FISH80] that, given a position p and window (a,b), f = falphabeta(p,a,b) obeys the following relation:

1. if $f \leq a$, negamax(p) $\leq$ f
2. if $f \geq b$, negamax(p) $\geq$ f
3. if $a < f < b$, negamax(p) = f

Thus a search that, say, fails high can use (f,INFINITY) as the window for the second search (rather than (b,INFINITY)). Falphabeta is guaranteed to search the same nodes as alphabeta, and the only overhead of falphabeta is the 'max' operation in the recursive procedure call.

The most obvious application of aspiration searching has already been mentioned, namely guessing an initial window and repeating the search in case of failure high or low. Though this method is more effective if there is some prior knowledge about the score distribution, it is applicable even in the absence of such information. Aspiration searching can benefit from the tighter bounds returned by falphabeta.

In the development of further optimizations of alpha-beta, the concept of a *minimal window* [FISH80] was

introduced. Assuming scores can only take integer values, (m,m+1) is an example of a minimal window. With such a window, the search will necessarily fail high or low. Though the true score of a position p cannot be found by a minimal window search, it does provide a bound on the score (i.e. determines whether or not negamax(p) > m), while making cutoffs that a full window search cannot. For example, if the score of one successor of p is found to exceed a bound m, an immediate cutoff can occur without searching the remaining successors of p for the one that exceeds m by the most. In many circumstances a bound of this type on a position is sufficient.

Lalphabeta [FISH80], for "last-move-with-minimal-window alphabeta", is one application of this concept. The last successor of the root node is compared with the bound m, where m is the best score found so far. If the search fails low, the previously established best move still applies. If the search fails high the last move has been determined to be best, though the precise score is not known. Figure 6 demonstrates this idea.

All the search algorithms discussed so far have been *directional*, i.e. there is some linear arrangement of the terminal nodes such that the algorithms never examine a node to the left of one previously examined [PEAR80]. All the remaining algorithms in this section are *non-directional*; no guarantee can be made about a 'left-to-right' examination of

```
falphabeta(position p, int α, int β)
{   int m, i, t, w;
    if (terminal(p))
        return(staticvalue(p));
                                /*                              */
    w = generate(p);            /*  determine successors        */
                                /*      p.1,...,p.w             */
    m = -INFINITY;
    for i = 1 to w do
    {   t = -falphabeta(p.i, -β, -max(m, α));
        if (t > m)  m = t;
        if (m ≥ β)  return(m);  /* cutoff */
    }
    return(m);
}
```

Figure 5: Falphabeta

```
lalphabeta(position p, int α, int β)
{   int m, i, t, w;
    if (terminal(p))
        return(staticvalue(p));
                                /*                              */
    w = generate(p);            /*  determine successors        */
                                /*      p.1,...,p.w             */
    m = α;
    for i = 1 to w-1 do
    {   t = -alphabeta(p.i, -β, -m);
        if (t > m)  m = t;
        if (m ≥ β)  return(m);  /* cutoff */
    }
    t = -alphabeta(p.w, -m-1, -m);
    if (t > m)  m = t;
    return(m);
}
```

Figure 6: Lalphabeta

the terminal nodes.

Palphabeta, for "principal-variation alphabeta" [FISH80], is a generalized application of minimal window searching. It can, under certain circumstances, (re)examine nodes to the left of others already scored. If the first path to a terminal node is in fact the optimal sequence of moves predicted by minimax, the balance of the tree is searched with a minimal window. However each time a minimal window search on a subtree fails high, the search is repeated with a wider window. Hence there is some risk, if the tree is poorly ordered, that palphabeta will visit more terminal nodes than alphabeta. There exist certain techniques, particularly *iterative deepening* (see Appendix 1), which can provide a prefix to the actual principal variation with reasonable reliability. Such techniques increase the feasibility of palphabeta. The palphabeta algorithm is given in Figure 7.

SCOUT [PEAR80] is a further generalization of palphabeta, where instead of calling alphabeta after a minimal window search fails high, a recursive call is is made to SCOUT. As in palphabeta, the possibility for reexamining nodes in this case makes SCOUT non-directional. Figure 8 gives an adaptation of SCOUT, reformulated into the negamax approach.

```
palphabeta(position p)
{   int m, i, t, w;
    if (terminal(p))
        return(staticvalue(p));
                                /*                              */
    w = generate(p);            /*  determine successors        */
                                /*     p.1,...,p.w              */
    m = -palphabeta(p.1);
    for i = 2 to w do
    {   t = -falphabeta(p.i, -m-1, -m);
        if (t > m)
            m = -alphabeta(p.i, -INFINITY, -t);
    }
    return(m);
}
```

Figure 7: Palphabeta

```
SCOUT(position p)
{   int m, i, t, w;
    if (terminal(p))
        return(staticvalue(p));
                                /*                              */
    w = generate(p);            /*  determine successors        */
                                /*     p.1,...,p.w              */
    m = -SCOUT(p.1);
    for i = 2 to w do
    {   t = -alphabeta(p.i, -m-1, -m);
        if (t > m)
            m = -SCOUT(p.i);
    }
    return(m);
}
```

Figure 8: SCOUT

The original version of SCOUT did not employ the minimal window idea, but rather a similar procedure, TEST [PEAR80], which is applicable to both continuous and discrete score distributions. TEST could be used instead of the minimal window call to alphabeta in the body of the SCOUT procedure. TEST is given in Figure 9, also reformulated into the negamax framework. A potential disadvantage of SCOUT is that, unlike palphabeta, the bound that is returned by TEST (or a minimal window search) is not used to further reduce search.

SSS* [STOC79] is a non-directional algorithm for determining the minimax value of AND/OR trees, of which game trees are a special case. It is claimed that SSS* *dominates* alpha-beta in terms of terminal nodes evaluations, i.e. SSS* never scores a node that alpha-beta can ignore [STOC79]. However Appendix 3 provides a simple counter-example to this claim, and describes a modification to SSS* which guarantees its domination over alpha-beta. The modified version of SSS* will be used in the comparative studies in this thesis.

SSS*, for practical score distributions, can be shown to evaluate strictly fewer nodes than alpha-beta [STOC79]. This is achieved by means of a very large data structure, called the *OPEN list*, which simultaneously maintains a number of alternate solution paths throughout the tree. In uniform game trees of depth d and width w, the OPEN list is of order w**d/2 elements.

```
TEST(position p, int v)
{   int i, w;
    if (terminal(p))
        if (staticvalue(p) > v)
            return(TRUE);
        else
            return(FALSE);
                                /*                              */
    w = generate(p);            /*  determine successors        */
                                /*      p.1,...,p.w             */
    for i = 1 to w do
    {   if (not(TEST(p.i, -v)))
            return(TRUE);
    }
    return(FALSE);
}
```

Figure 9: TEST

To further improve searching performance, SSS* can draw upon the aspiration search idea used in alpha-beta. Assume SSS* is given a window (a,b) over which to conduct the search. The use of the lower bound is trivial; if at any point the top item in the OPEN list has score h $\leq$ a, the search can be terminated, returning h as an upper bound on the actual position score. The use of the upper bound b is more interesting. Instead of initializing the OPEN list to (ROOT,LIVE,+INFINITY), (ROOT,LIVE,b) is used. For an example of how this can reduce search, see the example in Appendix 3. Aspiration SSS* can also be employed as a bound testing procedure, similar to TEST, by the use of a minimal window.

SSS* performs particularly well, relative to alpha-beta, when the actual principal variation is to the right of the game tree. However in trees that are strongly ordered, the advantage is diminished. Since the SSS* algorithm is more time consuming than alpha-beta (for example, there is an insert into an ordered list), SSS* may only be practical if there is reason to believe that it will substantially outperform alpha-beta, and even then the storage requirements limit the size of the tree examined.

Because of these problems, Stockman suggested that an amalgamation of SSS* and alpha-beta may be more practical. A first possibility would employ alpha-beta at the top levels of the tree, with SSS* used to reduce terminal node evaluations at deeper levels. An important point to note is

that aspiration SSS* is very useful in such an algorithm. The nodes at the maximum search depth for alpha-beta will have a window which can be utilized by SSS* to advantage. An alternative would employ SSS* at top levels of the tree, with alpha-beta used to search deeper in the tree. Though it might appear that this method is better than alpha-beta/SSS*, particularly for random trees, there is a serious disadvantage present. This arises from the fact that SSS* has no lower bound on a node score comparable to the alpha value in the alpha-beta algorithm. Thus when alpha-beta is called from SSS*, alpha must be initialized to -INFINITY to guarantee success. The effects of this will be seen in the empirical studies. However, both alpha-beta/SSS* hybrids can significantly reduce storage requirements.

Another variation on SSS* is proposed which would employ the algorithm to some fixed depth d, whereupon the 'terminal nodes' at d ply could be evaluated by a further depth d SSS* search. The staging reduces storage requirements so that they are linear with search depth. This approach uses SSS* in layers, or stages, and will be called *staged SSS**. As above, staged SSS* suffers from the fact that no lower bound is available on any given nodes score.

2.2 Performance Comparison

There are a number of alternative methods to measure the performance of algorithms that search game trees. These measures and their relative merits are examined here.

Elapsed CPU time is excellent as a performance measure provided (1) a comparison between algorithms is not attempted, and (2) the values are used only to determine relative performance. In other words, this measure is most useful when comparing the relative performance of a single algorithm on different types or sizes of trees. Relaxing the above restrictions reduces the validity of the measure. Comparing two different algorithms in this way is highly dependent upon the relative efficiency of the machine language encodings of the algorithms. In addition, the values are very machine dependent. It should be noted, however, that elapsed CPU time is the only performance measure discussed here which captures the idea that a more time-consuming algorithm is less desirable than a faster one, all else being equal.

NBP, for *Number of Bottom Positions* is a common measure of search performance [SLAG69]. By simply counting the number of terminal node evaluations, NBP provides a means of comparing algorithms' performances on trees of practical sizes. However NBP does not measure the amount of processing that must be done in order to choose nodes for evaluation,

and makes the implicit assumption that terminal evaluations are the major cost in a tree search.

Total Nodes Visited is similar to NBP except it includes the cost of non-terminal nodes as well. 'Total nodes visited' is rarely used as a performance metric for sequential programs, as it has a very similar character to the more easily calculated NBP.

The **asymptotic branching factor** as a cost measure can be defined as follows [BAUD78]:

> If N(w,d) is the number of terminal nodes examined by some algorithm A in searching a uniform tree of branching factor w and depth d, then
>
> ```
> lim (N(w,d)**1/d)
> d->oo
> ```
>
> is called the *branching factor of algorithm A.*

This cost measure often has little practical application, as depths of trees necessary to display the asymptotic properties are computationally infeasible to search.

It is clear that CPU time is a very good performance measure for practical systems, since minimizing the real search time is often the main goal. However theoretical and empirical studies rarely use this measure, since it is so dependent on the application and the efficiency of the programs. Theoretical studies have concentrated on NBP and

branching factor as performance measures, while empirical studies usually measure NBP.

The theoretical performance characteristics of some of the previously discussed algorithms are now examined. For purposes of analysis it is convenient to compare searching performance on uniform game trees of depth d and width w. It is further assumed that terminal nodes are assigned discrete values.

The negamax algorithm visits all the nodes in a game tree, and in particular all w**d terminal nodes. Thus the branching factor of negamax is w. The performance of the remaining algorithms is dependent on the ordering properties of the tree being considered. It has been shown that any tree searching algorithm must examine at least

$$w^{\lceil d/2 \rceil} + w^{\lfloor d/2 \rfloor} - 1$$

terminal nodes [SLAG69]. Alpha-beta, under optimal ordering conditions, can attain this lower bound. On the other hand, the worst case performance of alpha-beta matches minimax. The branching factor of alpha-beta has been shown to be w**1/2 (with some exceptions) [PEAR80]. This is optimal over all searching algorithms.

Palphabeta, SCOUT, and SSS* can also achieve the lower bound on NBP. It has also been shown that SCOUT attains the optimal branching factor of w**1/2 [PEAR80]. Since SSS*

examines fewer nodes than alpha-beta, SSS* must also have branching factor w**1/2.

Aspiration searching has been examined theoretically [BAUD78], and it is shown that for trees with typical game-playing characteristics, a speedup of between 15% and 25% can be expected. In other words, aspiration alpha-beta is, under normal circmstances, better than alpha-beta.

In trees in which the w**d terminal nodes are independent identically distributed random variables with a continous distribution function, a general formula for the average number of terminal positions scored by alpha-beta has been developed [FULL73]. This formula is computationally intractable however, and can only be calculated for small values of w and d.

For trees of practical depths the branching factor of an algorithm is not an adequate performance measure, and NBP is a more relevant means of comparing different search methods. At present, if it is desired to study searching performance on trees with varying types of ordering properties, only empirical methods are available. In the following study a number of algorithms will be compared:

a. alphabeta (AB)

b. palphabeta (PAB)

c. SCOUT (SC)

d. SSS* (SSS)

e. staged SSS* (SS)

f. SSS*/alphabeta hybrid (SAB)

g. alphabeta/SSS* hybrid (ABS)

Lalphabeta, strictly speaking, is not as powerful as the other algorithms since the position score is not always determined, and will not be included here. Studies on checkers game trees [FISH80] indicate that lalphabeta can produce a minimal savings over alphabeta (about 1.5%).

The trees to be searched will be uniform with (width,depth) combinations: (8,2), (16,2), (24,2), (8,4), (16,4), (24,4) and (8,6). It is also desired for the trees to range over various types of ordering properties. The different sizes and ordering of the trees should give some indication of the strengths and weaknesses of the algorithms. In this study, trees are classified by the distribution of the placing of the best move at any given node. The following distributions are employed:

a. random

b. .5 first-move-best (moderately ordered)

c. geometric with parameter .5 (moderately ordered)

d. .8 first-move-best (strongly ordered)

e. geometric with parameter .8 (strongly ordered)

f. best-first (perfect) ordering

By studying trees with the above properties, it is hoped to compare algorithms in situations of practical interest. Current chess programs are able, using the search

enhancements discussed in Appendix 1, to approach optimal ordering in their trees. Thus the emphasis placed on strongly ordered trees. In Appendix 2, complete details are given on the mechanisms used for tree generation.

The values in the following tables were determined by the independent generation of 100 trees of the desired properties, and searching them with each applicable algorithm. The average number of terminal nodes for each search method is recorded in the tables. 100 was chosen as the sample size for practical reasons, as the generation of the larger trees requires considerable CPU time.

Tables 1, 2, and 3 contain 2 ply data. Only alphabeta, palphabeta, SCOUT, and SSS* are compared, since staged SSS* and alpha-beta/SSS* hybrids are not applicable for 2 ply trees. A number of observations can be made. Firstly, all the algorithms are able to attain the minimum number of evaluations on optimally ordered trees. Although not apparent from the data presented, in fact SSS* dominates alphabeta dominates palphabeta dominates SCOUT. This could have been predicted beforehand. SSS* is known to dominate alpha-beta. Both palphabeta and SCOUT, when carrying out their minimal window searches, repeat the search on failure high. The place where palphabeta and SCOUT try to compensate for this repetition, the failure low minimal window searches, evaluate exactly the same nodes as alphabeta in two ply trees. Thus alphabeta dominates palphabeta and

	rand	.5	.5 g	.8	.8 g	opt.
AB	34.4	26.9	23.7	19.1	17.7	15
PAB	43.3	31.4	26.0	20.1	18.1	15
SC	51.5	37.0	31.1	22.3	19.5	15
SSS	24.9	21.0	18.8	16.9	16.3	15

Table 1: Tree width=8, depth=2

	rand	.5	.5 g	.8	.8 g	opt.
AB	89.7	64.6	49.5	44.2	37.0	31
PAB	108.9	70.9	51.8	45.7	37.4	31
SC	128.6	86.1	62.7	52.6	41.0	31
SSS	62.2	47.3	38.8	36.8	33.4	31

Table 2: Tree width=16, depth=2

	rand	.5	.5 g	.8	.8 g	opt.
AB	161.8	98.1	75.2	67.8	54.6	47
PAB	191.7	107.0	77.4	69.9	55.1	47
SC	234.7	127.9	94.2	79.3	59.0	47
SSS	100.7	73.9	58.6	57.7	50.7	47

Table 3: Tree width=24, depth=2

SCOUT. Palphabeta dominates SCOUT because, after a failing high search, palphabeta examines all the terminal nodes of the given subtree once more, but each node that is better than its earlier siblings is examined twice more by SCOUT. In addition, palphabeta is sometimes able to use the tighter bound returned by the initial call to falphabeta to cutoff search earlier.

Comparing the values for alphabeta on random trees with those calculated by Fuller's formula [NEWB77], they are found to be consistently lower. This is not unexpected, since Fuller's formula assumes distinct values for all terminal nodes. When the values are chosen from a discrete distribution, a search reduction is possible owing to the fact that cutoffs occur on equal scores.

Tables 4,5 and 6 contain data from 4 ply trees. It is now possible to include staged SSS* and SSS*/alpha-beta hybrids. The SS algorithm used here has stage depth 2. SAB uses a two ply SSS* search on top of a two ply alpha-beta search, while ABS does the opposite. It is apparent from the data that neither SS nor SAB are able to attain the theoretical minimum NBP on perfectly ordered trees. This results from the previously mentioned fact that SSS* does not maintain a lower bound for position scores.

Some dominations can be shown to exist at 4 ply:

1. SSS dominates AB

	rand	.5	.5 g	.8	.8 g	opt.
AB	712.3	392.2	306.1	234.6	183.1	127
PAB	757.1	383.0	299.5	229.6	178.0	127
SC	933.0	433.3	342.0	263.2	189.7	127
SSS	410.0	281.8	223.7	180.9	156.8	127
SS	493.8	352.5	282.9	240.5	208.1	176
SAB	709.4	461.2	353.9	281.0	229.7	176
ABS	576.5	340.4	272.2	216.3	172.5	127

Table 4: Tree width=8, depth=4

	rand	.5	.5 g	.8	.8 g	opt.
AB	4136	1936	1267	954	740	511
PAB	4043	1868	1219	895	715	511
SC	4711	2081	1280	939	739	511
SSS	2233	1249	921	779	633	511
SS	2789	1712	1207	1109	873	736
SAB	4391	2383	1538	1318	972	736
ABS	3254	1641	1118	874	697	511

Table 5: Tree width=16, depth=4

	rand	.5	.5 g	.8	.8 g	opt.
AB	10822	4883	2819	2320	1686	1151
PAB	10132	4574	2632	2210	1625	1151
SC	11434	4899	2690	2337	1692	1151
SSS	5690	3087	2066	1770	1424	1151
SS	7275	4385	2741	2659	1987	1680
SAB	11884	6132	3558	3186	2224	1680
ABS	8420	4104	2460	2110	1591	1151

Table 6: Tree width=24, depth=4

	rand	.5	.5 g	.8	.8 g	opt.
AB	12452	5299	3727	2298	1813	1023
PAB	11665	4775	3353	2083	1654	1023
SC	14370	5205	3581	2160	1679	1023
SSS	6178	3296	2492	1774	1501	1023
SS	9182	5385	3988	3102	2528	1856
SA2	14033	6813	4684	3234	2500	1464
SA4	11481	5843	4212	2901	2300	1464
AS2	8590	4065	3023	1980	1632	1023
AS4	10762	4757	3408	2160	1729	1023

Table 7: Tree width=8, depth=6

2. SSS dominates SS
3. SSS dominates ABS
4. ABS dominates AB
5. SS dominates SAB

Some interesting cases of non-domination:

1. PAB does not dominate SC
2. SSS does not dominate SC or PAB
3. SS does not dominate ABS, and vice versa

Examining the values in the tables, some conclusions can be drawn. For random trees, SSS, SS and ABS are noticeably superior in terms of NBP. On trees of width 24, SSS is 55% of AB, SS is 67% of AB, and ABS is 78% of AB. The standard deviations were:

a. SSS - 843
b. SS - 815
c. ABS - 1737

From this data it appears that SSS* and staged SSS* should be used for search of random trees. They perform well, with a relatively small standard deviation. SSS* requires OPEN list storage of 24**2 = 576 entries, while staged SSS* only needs 2*24 = 48 entries.

As the trees become better ordered, PAB and SC begin to deserve consideration. For width 24 trees both methods are slightly better than AB, and under favorable conditions can beat SSS*. Both methods have a relatively large standard

deviation compared to AB (524 vs. 912 and 961 are the width 24 values). ABS and SSS remain good choices for ordered trees from a node evaluation point of view. However their advantage over AB is proportionally less, and the more time-consuming algorithms become less attractive.

Some further tests were conducted, comparing palphabeta and SCOUT with alphabeta on trees which had the correct principal variation but were random otherwise. PAB and SC searched the same nodes, while AB searched about 40% more nodes (on trees of depth 4 and width 8).

The 6 ply data provides further support for the conclusions drawn from the 4 ply data. The algorithms tested here included:

1. SS - staged SSS* with 3 stages of depth 2.
2. SA2 - 2 ply SSS* over 4 ply alphabeta.
3. SA4 - 4 ply SSS* over 2 ply alphabeta.
4. AS2 - 2 ply alphabeta over 4 ply SSS*.
5. AS4 - 4 ply alphabeta over 2 ply SSS*.

Again SSS*, staged SSS* and alphabeta/SSS* show well on random trees, with palphabeta and SCOUT becoming relatively effective on ordered trees.

A final test was done to compare aspiration SSS* with aspiration alpha-beta. The test was run on random trees of depth 4 and width 8, with the tree score set to 64. SSS* was given an upper bound of 65, while alphabeta was tested with

the window (63,65). It was found that in almost all cases identical nodes were examined. There were some exceptions where SSS* was able to make additional cutoffs when a particular node's score was equal to 64. Since this score can be within the alpha-beta window initially, alphabeta cannot make this cutoff.

3. Approaches to Parallel Tree Search

There are a number of methods for applying parallelism to game tree search. Though this thesis is primarily concerned with tree decomposition methods, some other possibilities are mentioned for the sake of completeness.

3.1 Parallelism in Primitive Operations

Two basic operations of most programs that search game trees are *move generation* and *terminal node evaluation*. Both these functions are promising sites for special purpose multi-processors, particularly in chess. Parallel chess move generation is discussed in [CORA76], and parallel evaluation in [MARS80]. It is important to note that cooperation between processors is occurring at a very low level, likely requiring a highly specialized interconnection mechanism.

3.2 Parallel Aspiration Searching

The basis of aspiration searching is the improved performance of the alpha-beta algorithm on a restricted window. Aspiration searching has a parallel counterpart, i.e. searching a number of (disjoint) windows simultaneously. The advantage of this method is that the concurrent searches are relatively independent, reducing the need for a complex communication scheme.

Decomposing the window into suitable sub-intervals has been investigated [BAUD78]. Rather than choosing the intervals such that their union is (-INFINITY,+INFINITY), it has been found that leaving 'gaps' introduces the possibility for feedback between the concurrent searches. Optimal decompositions are usually of this form.

The main difficulty with the parallel aspiration approach is that the overall search time is bounded below by the search time for alpha-beta under optimal ordering conditions, i.e. there is a minimal tree that must be examined in any successful search. Therefore, regardless of the number of processors available, there is a fixed maximum speedup possible. A typical bound on speedup is a factor of five or six [BAUD78]. However, when the number of cooperating processors is small (k=2 or 3), a speedup of more than k is possible. This result shows that sequential aspiration searching (which can simulate the k=2 or 3 case easily enough) is better than standard alpha-beta [BAUD78].

In any parallel searching algorithm using the window concept, parallel aspiration search is applicable. The discussion of parallel algorithms in Chapter 4 will omit mention of parallel aspiration search, on the understanding that it is an additional enhancement which can be used where applicable.

3.3 Tree Decomposition

Most discussions of parallel game tree search have concentrated on concurrent examination of independent subtrees. Even Baudet concludes that parallel aspiration searching must be combined with tree decomposition if large performance improvements are desired [BAUD78]. However there are a number of overheads involved in concurrent search of different subtrees. These overheads can be divided into two broad categories, namely *search overhead* and *communication overhead*.

The efficiency of most search algorithms arises from the fact that decisions to cutoff search on given subtrees are based on all the accumulated information obtained to that point in the search. For various reasons, this information is not always available to parallel search algorithms. Communication delays may make the data arrive too late, or, more importantly, information may not yet be available as it is being calculated by another concurrent search. The extra effort that a given parallel algorithm must carry out (relative to the sequential algorithm) can be defined as the *search overhead*. It is possible to define the *search overhead coefficient* in an attempt to obtain a numerical measure of searching overhead.

Given parallel algorithm A(k), where k is the number of processors used, and game tree T, let N(A(k),T) be the number of terminal nodes scored by algorithm A(k) when

searching tree T. Then

$$S(A(k),T) = N(A(k),T)/N(A(1),T)$$

is called the *search overhead coefficient* of algorithm A(K) on tree T.

Note, in general, one would expect $S > 1$ for $k > 1$, though this is certainly not always the case. The quantity S provides an indication of how efficient a searching algorithm is in distributing information dynamically among the cooperating processors on a particular game tree.

Communication overhead can arise in different ways, depending on the system configuration. Information can be communicated via some sort of message passing system, or through a global shared data structure. The former incurs message passing costs, while the latter will require synchronization overhead, which increases with the degree of concurrency. Of course the volume of information to be shared is dependent upon the particular search algorithm used, but it seems clear that, in general, communication overhead is inversely related to search overhead. In other words, if improved sharing of data between independent searches is achieved (at increased communication costs), better cutoff decisions can be made by the search algorithm, thus reducing search overhead.

4. Algorithms for Parallel Search

Before discussing parallel search algorithms, it is necessary to make some assumption about the underlying processor architecture. Tree searching multi-processor systems can be classified into two basic categories, depending upon how they decompose trees for concurrent search. *Static decomposition* systems generate and assign subtree searches in a fixed, pre-determined manner, while *dynamic decomposition* systems assign subtree searches conditional on the current status of the overall search.

An architecture suitable for static decomposition is the *processor tree* [FISH80]. A processor tree consists of processors (the nodes of the tree) and communication lines (the branches of the tree). The successors of a node are its *slaves*, while the predecessor of a node is its *master*. There is one processor with no masters, the *root* processor. From this description it is clear that a given processor can communicate directly only with its master and slaves (if any).

The processor tree architecture is an excellent one from the implementation point of view. There are limited interconnection requirements for each processor, independent of the total number of processors in the system. Also, the number of processors is extendable in a simple and regular fashion, by increasing the width and/or depth of the tree. The processor tree also provides a fairly flexible means to

control the subtrees searched. If, for example, a master processor wants a sub-subtree to be evaluated, it can simply assign one of its slaves (and thereby all its descendants) to the search.

An architecture that can be utilized by a dynamic decomposition system has been suggested [AKL80]. It relies on a pool of processors selecting and running processes from a priority ordered set. Dynamic decomposition can reduce the idle time problem substantially, and provides the maximum possible flexibility in directing search towards specific desired subtrees. On the other hand, such a processor pool causes a number of implementation difficulties. In particular, a means of selecting and suspending processes must be found, one which does not involve inordinate synchronization and storage overheads. In addition, an efficient scheme must be devised which would allow an information update to currently running processors, a relatively trivial matter in the processor tree architecture.

The algorithms that will be described in the remainder of this chapter are based on a processor tree, static decomposition architecture. This choice was made in order to increase the practicality of the algorithms.

4.1 Tree-splitting

One means of implementing the alpha-beta algorithm on a processor tree is called the *tree-splitting algorithm* [FISH80]. In this algorithm, a master processor will generate all the successors of a given position, and assign them to its slave processors. Terminal slaves will carry out a regular alphabeta search on their assigned position, while non-terminal slaves will again generate and assign successors. Master processors maintain a local alpha-beta window, which they pass to their slaves along with a search assignment. The windows are updated when slaves return values from their searches.

This method of apply a processor tree does have some drawbacks. The width and depth of the tree are bounded by the width and depth of the game tree being searched. However, as will seen in this chapter, processor trees with large fanouts have great problems with search overhead. Therefore the tendency is to prefer deep, narrow tree structures to wide, shallow ones. For this reason, the maximum depth restriction is likely to be the more serious one.

Although processor trees are relatively powerful at directing search towards relevant game subtrees, there is some difficulty with processor idle time, since a given processors' descendants cannot be reassigned until the

initial search is completed. This idle time is directly related to the processor tree width.

Figure 10 illustrates the tree-splitting algorithm. Several constructs have been adapted from Fishburn [FISH81].

1. j.treesplit indicates the execution of procedure treesplit on processor j.
2. **parfor**, a parallel for loop, conceptually creates a separate process for each iteration of the loop. The program continues as a single process when all iterations are complete.
3. **when** waits until its associated condition is true before proceding with the body of the statement.
4. **critical** allows only one process at a time into the critical region.
5. procedure *terminate* kills all processes in the for loop that are still active.

There is also a means for dynamically updating the alpha-beta window in slaves while they are carrying out a search. Consider the updating algorithm in Figure 11. Assuming a master processor receives a new alpha value from one of its slaves, UPDATE is invoked (via an interrupt mechanism) in each of the slaves currently searching.

A naive application of the tree-splitting algorithm might use one master and K slaves, with the master generating all the positions at some fixed common depth C in

```
treesplit(position p, int α, int β)
{   int w, i, t[MAXWIDTH];
    processor j;
    if I am a leaf processor
        return(alphabeta(p,α,β));
                                /*                          */
    w = generate(p);            /*  determine successors    */
                                /*       p.1,...,p.w        */
    parfor i = 1 to w do
    {   when (a slave j is idle)
        {   t[i] = -j.treesplit(p.i,-β,-α);
            critical
            {   if (t[i] > α) α = t[i];
            }
            if (α ≥ β)
            {   terminate();
                return(α);
            }
        }
    }
    return(α);
}
```

Figure 10: Treesplit

```
int alpha[MAXDEPTH], beta[MAXDEPTH];
/*
  each terminal processor keeps its alpha and beta values
  in global arrays instead of passing as parameters
*/
UPDATE(depth, score, bound)
{   if (bound == -1)        /*  lower bound  */
        alpha[depth] = max(alpha[depth], score);
    else
        beta[depth] = min(beta[depth], score);
    if (depth < MAXDEPTH - 1)
        UPDATE(depth + 1, -score, -bound);
}
```

Figure 11: UPDATE

the tree and assigning them successively to the slaves. Though having the appeal of simplicity, there are a number of drawbacks to such a scheme, based mainly on the tradeoffs involved over the value of the common depth.

For example, if C = 1, i.e. the slaves are assigned the immediate successors of the root node,

a. The degree of concurrency is immediately limited by the branching factor of the game tree.

b. There can be difficulty with system idle time, e.g. 7 slave processors will perform only slightly better, on the average, than 4 when searching a tree with branching factor 8.

c. There may be poor bound sharing between searches, thus increasing search overhead.

This last point deserves further discussion. Consider a uniform game tree T of width 8 and depth 4 which is perfectly ordered. The search overhead coefficient will be examined for various processor tree widths, assuming one level of slaves with C = 1.

S(A(1),T) = 127/127 = 1.0

S(A(3),T) = 190/127 = 1.496

S(A(5),T) = 308/127 = 2.425

S(A(9),T) = 586/127 = 4.472

In other words, a system with 9 processors that uses this configuration has a speedup factor, for perfectly ordered trees, of only 1.61.

Increasing C, the common depth, postpones the limited concurrency problem, and reduces the difficulty with idle time, since the individual searches are shorter. In addition, search overhead is reduced considerably. The problems with increased C values are the greater communication overheads, and the increased complexity and volume of work required by the master processor. In fact, it is clear that the volume of work and the amount of storage needed for the master processor is exponential with C. This quickly provides a practical limit on the value for C.

In order to reduce the bottleneck at the master processor it is possible to insert some intermediate level masters between the root processor and the slaves searching at depth C. In this manner each master need only handle a fixed number of slaves, regardless of the total number of processors available. In such a configuration, define p to be the number of game tree plies between a master and its slaves. Then, given a processor tree of depth D, p * D = C, where C is again the depth at which the terminal slaves begin their search.

There are a number of variations on this technique designed to improve searching performance. If p is small (e.g. 1 or 2), the master processors could be idle much of the time waiting for messages. In this case, the masters may be able to join their slaves in subtree evaluation, although this is probably only practical for the deepest masters

[FISH80]. A second optimization could group higher level masters as separate processes on a single processor [FISH80]. The fact that the top level masters are usually the least busy motivates this suggestion, though the value of p again plays a similar role to above. A third variation on the tree-splitting algorithm involves a more complex processor assignment strategy, and is described in the next section.

4.2 Principal Variation Techniques

One parallel search algorithm [AKL80] was designed based on the observation that alpha-beta must search certain subtrees regardless of the ordering properties of the game tree. Thus these subtrees can advantageously be searched concurrently. However, the described algorithm is based on a dynamic tree decomposition, the disadvantages of which have already been discussed. *Mwf* [FISH81] is an adaptation of this method to the processor tree architecture.

Pv-splitting is an algorithm that relies on a somewat different assumption, i.e. the principal variation is correct. Pv-splitting, for "principal variation tree-splitting", is also based on a processor tree architecture. The algorithm can be motivated by a close examination of the behaviour of the sequential alpha-beta algorithm on perfectly ordered trees.

The Dewey decimal system will be used to assign coordinate numbers to nodes. Every position at depth k is represented by a sequence of k positive integers. The root is represented by a null sequence, while the w successors of a node a1.a2 ... ak are a1.a2 ...ak.1 through a1.a2 ... ak.w. It is now possible to precisely define perfect ordering. A tree is perfectly ordered if, for each position p in the tree, where p represents a sequence of integers,

negamax(p) = staticvalue(p) if p is terminal
= -negamax(p.1) otherwise.

The nodes visited by alpha-beta in a perfectly ordered tree are called *critical* nodes [KNUT75]. A node a1.a2 ... ak is critical if ai is 1 for all even values of i or all odd values of i. Critical nodes can be divided into three types. In type 1 nodes, all the ai's are 1. A node is of type 2 if aj is its first entry > 1 and k-j is even. When k-j is odd, the nodes are of type 3.

Intuitively, type 1 nodes are those on the principal variation, while type 2 nodes are alternatives to the principal variation. Successors of type 2 nodes are of type 3, while type 3 successors are again of type 2.

The following observations can be made about the critical positions in a perfectly ordered game tree:

a. At type 1 and 2 nodes, the best move must be

considered first, though this is not necessary for type 3 nodes.

b. At type 1 and 3 nodes, all successors are examined.

c. At type 2 nodes, only the first successor is examined.

Clearly the power of alpha-beta pruning derives from the fact that type 2 nodes can be cutoff with less than a full-width search. These cutoffs are made possible by the score returned from searching type 1 nodes. If a type 2 node, for example the second successor of the root, is searched without the benefit of the score from the corresponding type 1 node (in this case, the first successor of the root), the node will be explored full-width. A parallel algorithm must find a means to reduce this search overhead. The structure of the palphabeta algorithm suggests the algorithm shown in Figure 12, which is run on the root processor of a processor tree. This algorithm, pvsplit, concentrates its efforts on fully evaluating type 1 nodes, and then using the resultant score to search type 2 nodes efficiently. There must be some maximum depth that this procedure can be applied, due to the restriction that the processor tree should not be deeper than the game tree. At the maximum depth on the principal variation, a standard version of the tree-splitting algorithm may be used to obtain the initial score, after which pv-splitting can be

```
pvsplit(position p, int α, int β, int depth)
{   int w, i, t[MAXWIDTH];
    processor j;
    if (depth == MAXDEPTH)
        return(treesplit(p,α,β));
                                /*                          */
    w = generate(p);            /*  determine successors    */
                                /*      p.1,...,p.w         */
    α = -pvsplit(p.1, -β, -α, depth + 1);
    if (α ≥ β)
        return(α);
    parfor i = 2 to w do
    {   when (a slave j is idle)
        {   t[i] = -j.treesplit(p.i,-β,-α);
            critical
            {   if (t[i] > α) α = t[i];
            }
            if (α ≥ β)
            {   terminate();
                return(α);
            }
        }
    }
    return(α);
}
```

Figure 12: Pvsplit

used.

An optimization of pv-splitting arises from the observation that, in optimally ordered trees, type 3 nodes must be explored full-width, while type 2 nodes need only examine their first successor. This indicates that concurrency is more profitably applied at type 3 nodes, since no cutoffs can occur here. Type 2 nodes, on the other hand, should be examined with minimal concurrency, since a cutoff can occur after scoring only 1 successor. This technique is implemented in a processor tree by assigning slaves type 2 positions only, i.e. instead of assigning the immediate successors of a (type 2) node to the slaves, assign the successors' successors. Besides reducing search overhead, this optimization allows processor tree widths that are prohibitively expensive (in terms of search overhead) in standard tree splitting. Empirical justification can be found in Chapter 5.

So far it has been assumed that the tree being searched is perfectly ordered. Obviously if it is known in advance that the tree is perfectly ordered, there is no point in carrying out a search at all. Therefore as a practical matter, pv-splitting should be examined after relaxing the optimal-ordering assumption, though there is still good reason to assume the tree is strongly ordered, i.e. there is a high probability that the best move from a given position is placed high in the movelist. See Appendix 1 for the

applicability of standard sequential ordering techniques to a parallel environment.

Palphabeta again suggests a modification to the parallel searching algorithm, namely the use of the minimal window bound-testing procedure. If the type 1 nodes are indeed the correct principal variation, the remainder of the search can benefit from the minimal window. Whenever a minimal window search fails high, maximum effort should be made to fully evaluate this subtree, since it contains the new principal variation. In effect, the subtree should be treated as if its root was a type 1 node, since its value is crucial to the efficiency of the remaining searches. Chapter 5 examines the empirical performance of pv-splitting.

4.3 SSS* Adaptations

Although the effectiveness of sequential SSS* (in terms of NBP) cannot be disputed, a parallel version is fraught with implementation difficulties. These problems center around the maintenance of the requisite data structure. If the amount of storage required is not a limiting factor, the synchronization overhead involved in preserving the integrity of the data tends to reduce concurrency gains.

A parallel adaptation of SSS* can be envisioned which works in a dynamic decomposition framework similar to that described by Akl, i.e. processors choosing from a priority

ordered set of tasks. In this case, the tasks are individual entries in the OPEN list. A free processor can remove the top entry of the list and carry out the appropriate action, which will result in further additions or deletions to the list. If it were not for the inherent problems of a dynamic tree decomposition, this method could be very attractive in terms of overall search speed, particularly for trees that are randomly or poorly ordered.

Staged SSS* is adaptable to the processor tree architecture, and hence can employ a static game tree decomposition. Each master maintains an OPEN list, and assigns appropriate subtree searches to its slaves. Terminal slaves can employ either SSS*, staged SSS*, or any of a number of other methods to evaluate their assigned nodes. Since the various OPEN lists are local to a given processor, no data sharing overhead is required here. In addition, the total storage requirements are not exponential with game tree depth, making deeper searches more practical. Staged SSS*, judging from the sequential performance analysis, will be most useful in random or weakly ordered trees.

Similarly, the alpha-beta/SSS* hybrids can be adapted for implementation on a processor tree. Also possible are processor architectures that combine dynamic decomposition (for the SSS* phase) with static decomposition (for the alpha-beta phase).

5. Performance Comparison of Parallel Algorithms

The performance characteristics of some parallel tree searching algorithms are now examined. Empirical tests will be done over a range of multi-processor configurations and on trees of various types. All concurrency will be simulated. Only algorithms that employ a static tree decomposition will be considered, as the practicality of dynamic methods is not clear.

In measuring the performance of different algorithms on a given multi-processor system, the main concern is total elapsed time. If algorithm A is consistently faster than algorithm B, it is fair to say that A is better than B, regardless of the relative number of terminal node evaluations or move generations. For the purposes of this simulation study, elapsed time is broken in three components:

1. EVALTIME - time to evaluate a terminal node.
2. MVGENTIME - time to generate the moves at a non-terminal node
3. MESSAGETIME - time to pass a message between a master and slave, or vice versa

For purposes of comparison, it is assumed that EVALTIME is set to 1 time unit, while MVGENTIME and MESSAGETIME are negligible. While terminal node evaluation time is consistent over all algorithms, move generation mechanisms and message passing time can be algorithm dependent. It is

assumed these other overheads will have roughly the same relative magnitudes as the elapsed EVALTIME's, and thus can safely be ignored.

The algorithms to be compared here are:

1. tree-splitting (TS)
2. pv-splitting (PV)
3. staged SSS* (SSS)

Tables 8,9 and 10 contain the simulation results. Each processor-tree/algorithm combination searched 100 trees of width 24 and depth 4, and average elapsed times were recorded. Note that, on 4-ply trees, PV cannot employ a depth 3 processor tree, and SSS cannot use either a depth 2 or 3 system.

Table 8 indicates the superiority of staged SSS* on random trees. Both TS and PV consistently required about 50% more search time on identical multi-processor systems.

Table 9 contains data from trees with geometric distribution of the best move with parameter .8. PV was designed for such strongly ordered trees, and outperforms TS considerably, especially on the wider processor tree configurations. The depth 2 processor trees diminish the advantage somewhat. This is because the simulations only carried out pvsplit at the root processor of the tree. Depth 1 processors ran regular tree-splitting.

Processor (depth,width)	TS	PV	SSS
(1,2)	6443	6101	4305
(1,4)	4384	4028	2854
(1,8)	3273	2958	2032
(1,12)	3021	2644	1696
(2,2)	3689	3967	
(2,4)	1506	1973	
(2,8)	758	1273	
(3,2)	2317		
(3,4)	654		

Table 8: Search time, randomly ordered trees of width 24, depth 4

Processor (depth,width)	TS	PV	SSS
(1,2)	1264	944	1140
(1,4)	1187	604	690
(1,8)	1232	440	463
(1,12)	1270	412	383
(2,2)	766	686	
(2,4)	391	391	
(2,8)	236	287	
(3,2)	541		
(3,4)	244		

Table 9: Search time, strongly ordered trees of width 24, depth 4

Processor (depth,width)	TS	PV	SSS
(1,2)	863	599	876
(1,4)	719	311	497
(1,8)	647	167	284
(1,12)	623	119	213
(2,2)	575	443	
(2,4)	287	190	
(2,8)	143	83	
(3,2)	431		
(3,4)	179		

Table 10: Search time, optimally ordered tree of width 24, depth 4

SSS shows rather well in this data set, but it should be noted that the algorithm is slightly more time-consuming than TS or PV, and thus SSS must do better than these in order to be practical.

A very interesting result occurs in the TS data. It appears that a system with 8 terminal slaves does worse than a system with 4 terminal slaves. This strange result is due to the lack of a dynamic updating mechanism in the simulations. When a processor searching a sub-optimal move returns first with a poor score, it will be reassigned but with the returned score as the new alpha value. This poor alpha value allows very few cutoffs, and increases search times. Even if the best possible alpha value is returned 1 time unit later, it is not available for the other search. These results indicate the great importance of dynamic window updating.

Table 10 contains data from a perfectly ordered tree. Such trees are ideally suited for PV, and the data bears this out. The values are of little practical interest, but they do illustrate dramatically the larger processor tree widths feasible with PV.

A brief analysis of the effectiveness of the parallelism is now presented. The speedups presented are determined by least squares fitting on the log-log graph of time vs. number of terminal processors. Assuming k terminal

processors, SSS achieves speedups of k**.52 (random ordering), k**.61 (strong ordering), and k**.8 (optimal ordering). However varying the methods used for choosing the next subtree to search can affect this performance. The particular algorithm employed in the simulations was favorable for ordered trees. Other mechanisms could be expected to favor random trees.

TS is best examined on processor trees of small fixed width so as to reduce the effects of the absence of a dynamic updating mechanism. The following values assume a processor tree width of 2. Randomly ordered trees gave a speedup of k**.74. Strong ordering produced a speedup of k**.61, while optimal ordering gives a speedup of k**.5. These results are consistent with theoretical studies [FISH81], which predict a k**.5 speedup for optimal ordering, with increasingly effective use of parallelism as trees become less ordered. With dynamic updating, the non-optimal ordering speedups can be expected to improve.

Since pv-splitting was not employed in full generality for processor trees of depth 2, the depth 1 values will be used. However the non-optimal orderings have their values seriously affected by the lack of dynamic updating. Random trees allowed a speedup of k**.52. Strongly ordered trees produced a k**.55 speedup, and optimal ordering gave a speedup of k**.92. Only in the last case is the figure actually meaningful.

6. Conclusion

6.1 Summary of Results

The purpose of this thesis was to describe and compare algorithms for parallel search of game trees. The initial step towards this goal summarizes in Chapter 2 the currently available tree-searching algorithms. Some interesting results for sequential tree search were obtained here. The great similarity between minimal window searches [FISH80] and TEST [PEAR80] was pointed out. This, together with the reformulation of SCOUT into the negamax framework, outlines the similarities and essential difference between SCOUT and palphabeta. SSS* as originally presented [STOC79] is shown to do worse than alpha-beta under certain circumstances, and a minor modification to the algorithm is described which ensures domination over alpha-beta (Appendix 3). Aspiration searching, used effectively in the alpha-beta algorithm, is shown to be applicable to SSS* also. Alpha-beta/SSS* hybrids are examined critically, and staged SSS* is presented as a potentially useful search method. Empirical tests are done on a number of algorithms to indicate their effectiveness in various situations. The trees used for these tests are generated dependent upon the distribution of the placing of the best move at any given node.

Three approaches to parallel tree search are discussed in Chapter 3, and tree decomposition is identified as the

focus of this thesis. Chapter 4 describes tree-splitting [FISH80] as one implementation of alpha-beta in parallel. Pv-splitting is presented as a parallel search algorithm which is effective given certain ordering assumptions about the trees searched. A parallel adaptation of staged SSS* is also discussed as a potentially effective search method for random trees.

Chapter 5 compares the performance of these parallel algorithms, given widely ranging multi-processor configurations and tree characteristics. The apparent strengths and weaknesses of the various algorithms are discussed.

6.2 Topics for Future Research

This thesis suggests a number of areas for potential research. The sequential search algorithms discussed in Chapter 2 deserve to be tested more fully in an actual game playing program. Of particular interest is the relative performances in light of the search enhancement techniques described in Appendix 1.

Static decomposition methods are concentrated on in this thesis, mainly because of their advantages in actual implementation on a multi-processor. Dynamic decomposition has great promise if a means can be found to reduce the associated synchronization overhead.

The parallel algorithms described in Chapter 4 should be tested more thoroughly on a game playing multi-processor system. In this way a number of questions could be answered concerning time spent in the different phases of the tree search and interprocessor communication. The effect of search enhancement techniques on a parallel system is also a matter of interest.

References

AKL80 S. Akl, D. Barnard and R. Doran, "Design, analysis and implementation of a parallel alpha-beta algorithm", TR 80-98, Computing and Information Science Dept., Queen's University, Kingston, (1980).

BAUD78 G. Baudet, "The design and analysis of algorithms for asynchronous multiprocessors", Ph.D. thesis, Computer Science Dept., Carnegie-Mellon Univ. Pittsburgh, (1978).

CORA76 L. Coraor and J. Robinson, "Using parallel microprocessors in tree decision problems", Proceedings of the International Symposium on Mini and Micro Computers, IEEE, 51-55, (1976).

FISH80 J. Fishburn and R. Finkel, "Parallel alpha-beta search on Arachne", TR 394, Computer Science Dept., Univ. Wisconsin, Madison, (1980).

FISH81 J. Fishburn, "Analysis of speedup in distributed algorithms", Ph.D. thesis, TR 431, Computer Science Dept., Univ. Wisconsin, Madison, (1981).

FULL73 S. Fuller, J. Gaschnig and J. Gillogly, "Analysis of the alpha-beta pruning algorithm", Computer Science Dept., Carnegie-Mellon University, Pittsburgh, (1973).

GILL78 J. Gillogly, "Performance analysis of the Technology chess program", Computer Science Dept., Carnegie-Mellon University, Pittsburgh, (1978).

KNUT75 D. Knuth and R. Moore, "An analysis of alpha-beta pruning", *Artificial Intelligence 6*, 293-326, (1975).

MARS80 T.A. Marsland, M. Campbell and A. Rivera, "Parallel search of game trees", TR 80-7, Computing Science Dept., Univ. of Alberta, Edmonton, (1980).

NEWB77 M. Newborn, "The efficiency of the alpha-beta search in trees with branch dependent terminal node scores", *Artificial Intelligence 8*, 137-153, (1977).

PEAR80 J. Pearl, "Asymptotic properties of minimax trees and game-searching procedures", *Artificial Intelligence 14*, 113-138, (1980).

SLAG69 J. Slagle and J. Dixon, "Experiments with some programs that search game trees", *J. ACM 2*, 189-207, (1969).

STOC79 G. Stockman, "A minimax algorithm better than alpha-beta?", *Artificial Intelligence 12*, 179-196, (1979).

THOM81 K. Thompson, "Computer chess strength", To appear in 'Advances in Computer Chess 3', M.R.B. Clarke (ed.), Wiley, (1981).

Appendix 1 - Enhancements to Parallel Search[1]

Current sequential game playing programs have developed numerous move-ordering and search reduction techniques in order to improve the effectiveness of the alpha-beta algorithm. A summary of these modifications is presented, and their adaptability to parallel tree searching is examined. Many of the following techniques have been developed in efficiency-conscious full-width chess programs.

In carrying out a search of a chess game tree, it is not uncommon for positions to recur in numerous places throughout the tree. Rather than rebuild the subtrees associated with the transposed positions, it may be possible to simply retrieve the results stored in a table by a previous search. A **transposition table** is a large hash table, with each entry representing a position. Although there are many table management problems which must be solved, the technique has very low overhead for the large potential gains.

A minimal table entry could have the following format:

lock : move : score : flag : length : worth

lock - ensures the table position is identical to the tree position.

---

[1] This Appendix is based on work carried out by T.A. Marsland and this author in 1980-81.

move - best move in the position, determined from previous search.

score - previously computed subtree value.

flag - indicates whether *score* is upper bound, lower bound or true score.

length - number of plies in subtree that *score* is based on.

worth - used in table management, to select entries for deletion.

When a position reached during a search is located in the table (i.e. the *lock* matches), there are a number of possible actions:

1. If *length* is less than remaining length to be searched, *score* is ignored and the search is carried out as usual. However *move* is tried first in the position. The main advantage of this is that it saves a move generation, and also, since *move* has previously (in a shallower search) proven best, it is likely to be so again. Furthermore, *move* will direct the search toward positions that have been seen before, further increasing the effectiveness of the table.
2. If *length* ≥ remaining length to be searched
   a. if *score* is true score, this value is returned without further searching.
   b. otherwise, *score* is used to adjust the current alpha-beta bounds. This could either cause an immediate cutoff, or allow the search to continue

with a reduced window. If a search must be done, *move* will be tried first.

Transposition tables are most effective in chess endgames, where there are fewer pieces and more 'reversible' moves. Gains of a factor of 5 or more are typical, and in certain types of King and pawn endings, experiments with BLITZ[2] and BELLE[3] have produced trees of more than 30 ply, representing speedups of well over a hundred-fold. Even in complex middlegames, however, significant performance improvement is observed.

In a parallel environment, transposition tables continue to be effective, provided all the processors access the same table. The method is especially attractive since table usage is a naturally autonomous function, and can be partitioned for parallel execution. Furthermore, something useful can be done while waiting for access to the transposition table, namely proceed with the evaluation of the next subtree. If the position sought is not in the table, then no time is lost, otherwise the first result from either the tree recomputation or the table access is used.

Access delays to the transposition table can be reduced by dividing the table into ranges and providing a different processor for each partition. In any case, the table naturally splits itself into two portions, those positions

---

[2]R. Hyatt, A. Gower; U. of Southern Mississippi
[3]K. Thompson, J. Condon; Bell Telephone Laboratories

for white to move and those for black, Figure 13. This scheme is quite independent of the relationships between the game processors C1, C2 and C3, which share and provide updates for the transposition table memory. A potential bottleneck exists at processor P0, but this should not be severe since P0 has no significant computational functions, beyond those necessary for the routing operations.

The **killer heuristic** is based on the premise that if move **My** 'refutes' move **Mx**, it is more likely that **My** (the 'killer') will be effective in other positions. Any move which causes a cutoff at level N is said to have refuted the move at level N-1. There are many ways of using this information. For example, the program CHESS[4] maintains a short list of killers at each level in the tree, and attempts to apply them early in the search in the hope of producing a quick cutoff. A further advantage of the killer heuristic is that it tends to increase the usefulness of the transposition table. By continually suggesting the same moves, there is a greater possibility of reaching a position already in the table.

In its full generality, the killer heuristic can be used to *dynamically reorder* moves as the search progresses. For example, if a move **My** at level N refutes a move at level N-1, and **My** remains to be searched at level N-2, it is worth

---

[4]D. Slate, L. Atkin; Northwestern University

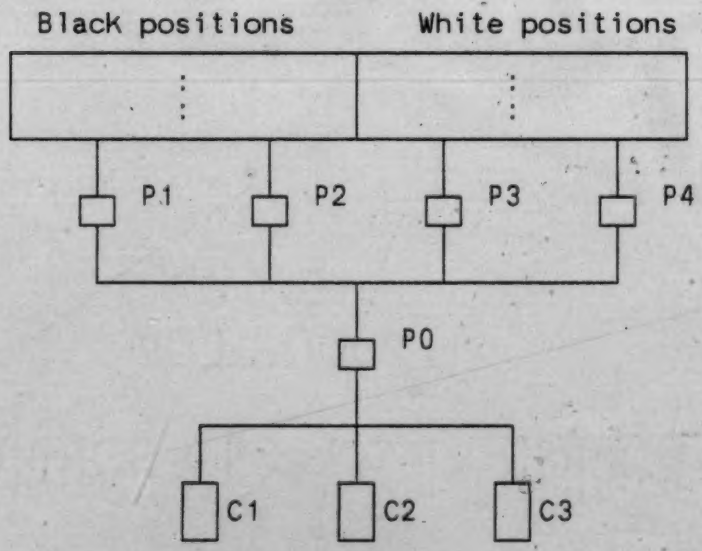

Figure 13: Transposition table access and management

considering next. An additional method, used by AWIT[5], seeks out defensive moves at ply N-1 which counteract killers from level N. The idea behind the generalized killer heuristic mechanism is to allow information gathered deep in the tree to be redistributed to shallower levels. This is not usually done by the full-width programs, however, since it is not yet clear that the potential gains exceed the overhead.

The killer heuristic presents similar problems to the transposition table when adapted to a parallel system. The killer list is so small, however, that the management problems are much reduced.

Iterative deepening refers to the procedure of using an N-1 ply search to prepare for an N ply search. This technique has certain immediately obvious advantages.

1. It can be used as a method of controlling the time spent in a search. In the simplest case, new iterations can be tried until a preset time threshold is passed.
2. An N-1 ply search can provide a principal continuation which with high probability contains a prefix of the N ply principal continuation. This allows the alpha-beta search to proceed more quickly.
3. The score returned from a N-1 ply search can be used as the center of an alpha-beta window for the N ply search. It is probable that this window will contain the N ply score, thus increasing search speed.

---

[5]T.A. Marsland; University of Alberta

These last two points, though significant, are not really complete justifications for the use of iterative deepening from a tree searching point of view. In fact, in experiments with checkers game trees [FISH80], it was found that iterative deepening increased the number of nodes searched by 20% (apparently only using point 2, however). In addition, studies with TECH[6] using a generalized version of point 2, but not 3, noted a 5% increase in search times when iterative deepening was applied [GILL78]. It appears that a strong initial move ordering, together with a good alpha-beta window estimate, can approximately match iterative deepening. The real **searching** advantage of iterative deepening is:

4. The transposition table and killer lists are filled with useful values and moves.

The importance of this fact is illustrated by the performance of the BELLE chess machine. Typical chess middlegame positions have branching factors of 35-40. It has been found that in such positions, it normally costs BELLE a factor of 5 - 6 to go one further ply, i.e. *less than the expected cost of optimal alpha-beta.*

A variation of this basic scheme, one which is especially appropriate if transposition tables are not used,

---

[6]J. Gillogly; Carnegie-Mellon University

is employed by L'EXCENTRIQUE[7]. A 2 or 4-ply minimax search is first performed to obtain W move-pairs (moves and their best refutation). These are then sorted and a 6, 8, 10 etc -ply iterative deepening cycle initiated. The rationale behind two ply increments is to preserve a consistent theme between iterations and thus discouraging a flip-flop of the principal variation between attacking and defensive lines.

Iterative deepening should be equally effective in sequential and parallel systems, though a parallel implementation must communicate partial principal variations between processors. Alternatively, if a transposition table is being used, the final principal variation can be obtained from there.

---

[7]C. Jarry; McGill University

Appendix 2 - Tree Generation Methods

For purposes of algorithm comparison it is desirable to have each algorithm search the same trees. For this reason it was decided to generate the trees separately and store them in files so as to allow them to be searched any number of times by any number of different algorithms.

A tree file is composed of w**d 8-bit bytes, with each byte representing the score of one terminal position. Only uniform trees of width w and depth d are considered. The position of a byte in the file indicates the location of a nodes in the game tree being represented. The first byte in the file is the leftmost terminal node in the tree, and the last byte the rightmost.

The terminal node values are chosen by a mechanism which depends upon the distribution of the position of the best move at a node. The tree is generated recursively by the procedure *gen*, of which a simplified version is shown in Figure 14. At any given non-terminal node, all the successors are assigned values less than **score**, and then the best move is chosen and assigned the value **score**. Note that gen works in the negamax framework. The initial call to gen would take the form

```
gen(ROOT,0,S)
```

where S can be either a fixed value or chosen randomly.

```
gen(position p, int depth, int score)
{   int value[WIDTH];
    int i;
    if (depth == MAXDEPTH)
    {   write(score);
        return;
    }
    if (odd(depth))
        for i = 0 to WIDTH-1 do
            value[i] = score - (rand() % (128+score));
    else
        for i = 0 to WIDTH-1 do
            value[i] = rand() % score;
    value[SelectBest()] = score;
                        /*                          */
    generate(p);        /*  determine successors    */
                        /*    p.1,...,p.WIDTH       */
    for i = 1 to WIDTH do
        gen(p.i, depth + 1, -value[i]);
}
```

Figure 14: Gen

Note: "%" is the modulus operator

For practical purposes it was found that trees with more than about 400,000 terminal nodes required excessive CPU resources, hence the size restrictions on the trees searched in Chapters 2 and 5.

Appendix 3 - The SSS* Algorithm

This appendix describes the SSS* algorithm as originally presented by Stockman, and gives an example where SSS* fails to dominate alpha-beta. A modification to SSS* is suggested which guarantees domination.

The following definitions are adapted from [STOC79]. SSS* is only described here in terms of game trees, though it is applicable to general AND/OR trees.

A *game tree* is an AND/OR tree whose root node is of type AND, all immediate successors of AND nodes are of type OR, and all immediate successors of OR nodes are of type AND.

A *solution tree* T of a game tree S is a tree with the following characteristics:

1. The root node of S is the root node of T.
2. If a non-terminal node of S is in T, then all of its immediate successors are in T if they are of type AND, and exactly one of its immediate successors is in T if they are of type OR.

The *value* of a solution tree T is denoted as f(T), and is defined as the minimum value of all terminal nodes in T.

It can be shown that for a solution tree T of game tree S, with root p,

minimax(p) ≥ f(T),

and, for some solution tree T0,

minimax(p) = f(T0).

Let T be a potential solution tree of game tree S. A *state of traversal* of T is a triple

(n,s,h)

where n is a node of T; s is status of solution of n and is either LIVE or SOLVED; and h is the merit of the state and an upper bound on f(T).

Now it is possible to give the SSS* algorithm:

(1) Place the start state (ROOT,LIVE,+INFINITY) on a list called OPEN.

(2) Remove from OPEN the state p=(n,s,h) with largest merit. Since OPEN is kept in non-decreasing order of merit, p is first in the list.

(3) If n = ROOT and s = SOLVED, terminate with h as the minimax value of the tree.

(4) Expand state p by applying state space operator G, described in Table 11.

(5) Go to (2)

The G operator requires the functions *type*, *first*, *next*, *parent and ancestor*, which are self-explanatory. Figure 15 gives an example of SSS* in operation, including the OPEN list at each application of the G operator (Figure 16). It should be noted that SSS* works in the minimax

Case of Operator G	Conditions satisfied by input state (n,s,h)	Action of G
1	s = SOLVED n ≠ ROOT type(n) = OR	Stack (m=parent(n),s,h) on list. Then purge OPEN of all states k where m is an ancestor of k in the game tree.
2	s = SOLVED n ≠ ROOT type(n) = AND next(n) ≠ NIL	Stack (next(n),LIVE,h) on OPEN list.
3	s = SOLVED n ≠ ROOT type(n) = AND next(n) = NIL	Stack (parent(n),s,h) on OPEN list.
4	s = LIVE n is terminal	Insert (n,SOLVED,min h,staticvalue(n)) on OPEN list behind all states of greater[1] or equal merit.
5	s = LIVE n is non-terminal type(first(n)) = AND	Stack(first(n),s,h) on OPEN list.
6	s = LIVE n is non-terminal type(first(n)) = OR	n = first(n) while n ≠ NIL do stack (n,s,h) on OPEN n = next(n)

Table 11: The G Operator

---

[1]In [STOC79], "lesser" is used, which requires an unnatural interpretation of "behind"

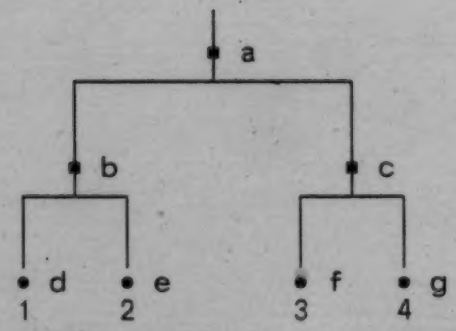

Figure 15: Tree to demonstrate SSS*

OPEN LIST -

1. (a,L,INF) #
2. (b,L,INF) (c,L,INF) #
3. (d,L,INF) (c,L,INF) #
4. (c,L,INF) (d,S,1) #
5. (f,L,INF) (d,S,1) #
6. (f,S,3) (d,S,1) #
7. (g,L,3) (d,S,1) #
8. (g,S,3) (d,S,1) #
9. (c,S,3) (d,S,1) #
10. (a,S,3) #

Figure 16: OPEN list for tree of Figure 15

framework, always evaluating terminal nodes from the viewpoint of one player.

The aspiration search idea used in alpha-beta is applicable to SSS* also. As an example of how a search reduction can take place, consider the subtree in Figure 17. Assume that an upper bound on the tree score of 5 is supplied. The OPEN list, Figure 18, at step 4 is

(f,LIVE,5) (g,LIVE,5) ... #

Applying the appropriate operator in SSS*, the list becomes

(f,SOLVED,min(5,7)) (g,LIVE,5) ... #

or

(f,SOLVED,5) (g,LIVE,5) ... #

The next step gives

(d,SOLVED,5) ... #

and node g is cutoff without evaluation. The intuitive justification of this is as follows: Since $f \geq 5$, and at d it is MAX to move, $d = \max\{5,g\} \Rightarrow d \geq 5$. This bound on d is sufficient to terminate search, since by the assumption that the score of the tree is less than 5, d cannot possibly be on the principal continuation, regardless of the value of f.

It can be shown that SSS* as described does not necessarily dominate alpha-beta when equal merit states are involved. Consider the tree in Figure 19. Alpha-beta will do terminal evaluations on nodes c, g, and h only. However figure 20 indicates that SSS* evaluates in addition nodes i and j.

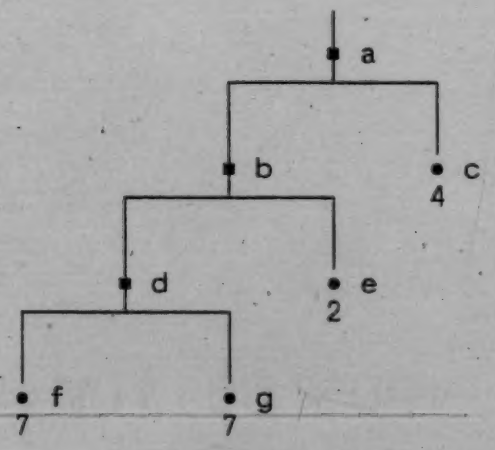

Figure 17: Tree for aspiration SSS*

OPEN LIST

```
 1.  (a,L,5) #
 2.  (b,L,5) (c,L,5) #
 3.  (d,L,5) (c,L,5) #
 4.  (f,L,5) (g,L,5) (c,L,5) #
 5.  (f,S,5) (g,L,5) (c,L,5) #
 6.  (d,S,5) (c,L,5) #     node g is cutoff
 7.  (e,L,5) (c,L,5) #
 8.  (c,L,5) (e,S,2) #
 9.  (c,S,4) (e,S,2) #
10.  (a,S,4) #
```

Figure 18: OPEN list for Figure 17

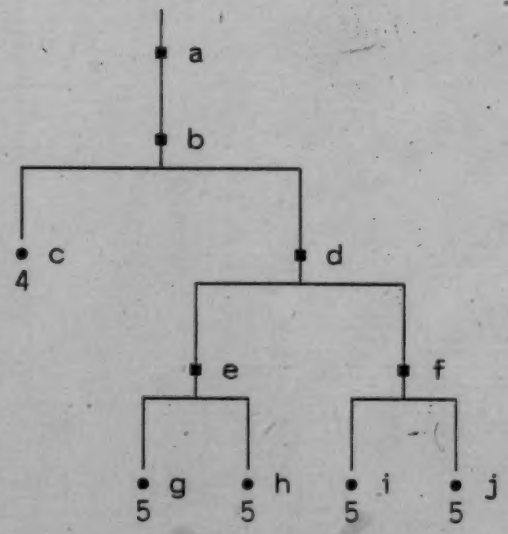

Figure 19: Tree illustrating non-dominance of SSS*

OPEN LIST

1. (a,L,INF) #
2. (b,L,INF) #
3. (c,L,INF) #
4. (c,S,4) #
5. (d,L,4) #
6. (f,L,4) (e,L,4) #
7. (i,L,4) (e,L,4) #
8. (e,L,4) (i,S,4) #
9. (g,L,4) (i,S,4) #
10. (i,S,4) (g,S,4) #
11. (j,L,4) (g,S,4) #
12. (g,S,4) (j,S,4) #
13. (h,L,4) (j,S,4) #
14. (j,S,4) (h,S,4) #
15. (f,S,4) (h,S,4) #
16. (d,S,4) #
17. (b,S,4) #
18. (a,S,4) #

Figure 20: OPEN list for tree of Figure 19

To guarantee that SSS* dominates alpha-beta, cases 4 and 6 of the G operator must be altered. These modifications will ensure that, when there are several states with equal merit, the *left-most* is always examined first. A node p is to the left of node q if, at their earliest common ancestor, p is in the subtree of a branch to the left of q. Table 8 gives the modified cases.

This modification produces other beneficial side effects (in addition to the fact that SSS* is ensured to dominate alpha-beta). Aspiration SSS* performs better if, given two subtrees of equal merit, one is evaluated fully before the other. In this way a fail high result can be found at less cost. A second result of the left-bias introduced here is improved searching performance if the tree being searched is strongly ordered, as is usually the case for practical applications of game tree search.

Case of Operator G	Conditions satisfied by input state (n,s,h)	Action of G
4	s = LIVE n is terminal	Insert (n,SOLVED,min h,staticvalue(n)) on OPEN list behind all states of greater merit and in front of all states of equal merit where n is to the left of the given node
6	s = LIVE n is non-terminal type(first(n)) = OR	n = last(n) while n ≠ NIL do stack (n,s,h) on OPEN n = previous(n)

Table 12: Modified Cases of the G Operator